Flume 1.3.0 User Guide

Introduction

Overview

Apache Flume is a distributed, reliable, and available system for efficiently collecting, aggregating and moving large amounts of log data from many different sources to a centralized data store.

Apache Flume is a top level project at the Apache Software Foundation. There are currently two release code lines available, versions 0.9.x and 1.x. This documentation applies to the 1.x codeline. Please click here for the Flume 0.9.x User Guide.

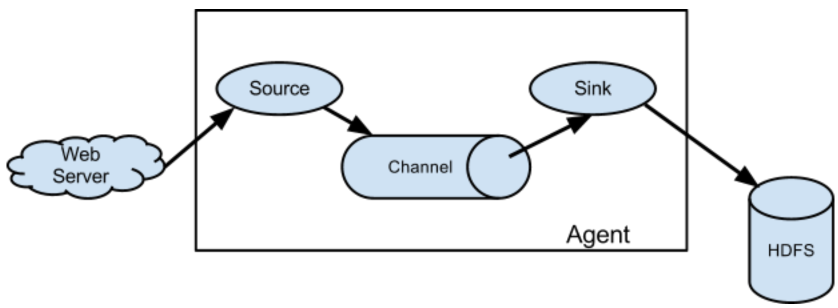
System Requirements

TBD

Architecture

Data flow model

A Flume event is defined as a unit of data flow having a byte payload and an optional set of string attributes. A Flume agent is a (JVM) process that hosts the components through which events flow from an external source to the next destination (hop).



A Flume source consumes events delivered to it by an external source like a web server. The external source sends events to Flume in a format that is recognized by the target Flume source. For example, an Avro Flume source can be used to receive Avro events from Avro clients or other Flume agents in the flow that send events from an Avro sink. When a Flume source receives an event, it stores it into one or more channels. The channel is a passive store that keeps the event until it's consumed by a Flume sink. The file channel is one example – it is backed by the local filesystem. The sink removes the event from the channel and puts it into an external repository like HDFS (via Flume HDFS sink) or forwards it to the Flume source of the next Flume agent (next hop) in the flow. The source and sink within the given agent run asynchronously with the events staged in the channel.

Complex flows

Flume allows a user to build multi-hop flows where events travel through multiple agents before reaching the final destination. It also allows fan-in and fan-out flows, contextual routing and backup routes (fail-over) for failed hops.

Reliability

The events are staged in a channel on each agent. The events are then delivered to the next agent or terminal repository (like HDFS) in the flow. The events are removed from a channel only after they are stored in the channel of next agent or in the terminal repository. This is a how the single-hop message delivery semantics in Flume provide end-to-end reliability of the flow.

Flume uses a transactional approach to guarantee the reliable delivery of the events. The sources and sinks encapsulate in a transaction the storage/retrieval, respectively, of the events placed in or provided by a transaction provided by the channel. This ensures that the set of events are reliably passed from point to point in the flow. In the case of a multi-hop flow, the sink from the previous hop and the source from the next hop both have their transactions running to ensure that the data is safely stored in the channel of the next hop.

Recoverability

The events are staged in the channel, which manages recovery from failure. Flume supports a durable file channel which is backed by the local file system. There's also a memory channel which simply stores the events in an in-memory queue, which is faster but any events still left in the memory channel when an agent process dies can't be recovered.

Setup

Setting up an agent

Flume agent configuration is stored in a local configuration file. This is a text file which has a format follows the Java properties file format. Configurations for one or more agents can be specified in the same configuration file. The configuration file includes properties of each source, sink and channel in an agent and how they are wired together to form data flows.

Configuring individual components

Each component (source, sink or channel) in the flow has a name, type, and set of properties that are specific to the type and instantiation. For example, an Avro source needs a hostname (or IP address) and a port number to receive data from. A memory channel can have max queue size (“capacity”), and an HDFS sink needs to know the file system URI, path to create files, frequency of file rotation (“hdfs.rollInterval”) etc. All such attributes of a component needs to be set in the properties file of the hosting Flume agent.

Wiring the pieces together

The agent needs to know what individual components to load and how they are connected in order to constitute the flow. This is done by listing the names of each of the sources, sinks and channels in the agent, and then specifying the connecting channel for each sink and source. For example, an agent flows events from an Avro source called avroWeb to HDFS sink hdfs-cluster1 via a file channel called file-channel. The configuration file will contain names of these components and file-channel as a shared channel for both avroWeb source and hdfs-cluster1 sink.

Starting an agent

An agent is started using a shell script called flume-ng which is located in the bin directory of the Flume distribution. You need to specify the agent name, the config directory, and the config file on the command line:

```
$ bin/flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties.template
```

Now the agent will start running source and sinks configured in the given properties file.

A simple example

Here, we give an example configuration file, describing a single-node Flume deployment. This configuration lets a user generate events and subsequently logs them to the console.

```
# example.conf: A single-node Flume configuration

# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Describe the sink
a1.sinks.k1.type = logger

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

This configuration defines a single agent named a1. a1 has a source that listens for data on port 44444, a channel that buffers event data in memory, and a sink that logs event data to the console. The configuration file names the various components, then describes their types and configuration parameters. A given configuration file might define several named agents; when a given Flume process is launched a flag is passed telling it which named agent to manifest.

Given this configuration file, we can start Flume as follows:

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.root.logger=INFO,console
```

Note that in a full deployment we would typically include one more option: `--conf=<conf-dir>`. The `<conf-dir>` directory would include a shell script *flume-env.sh* and potentially a log4j properties file. In this example, we pass a Java option to force Flume to log to the console and we go without a custom environment script.

From a separate terminal, we can then telnet port 44444 and send Flume an event:

```
$ telnet localhost 44444
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hello world! <ENTER>
OK
```

The original Flume terminal will output the event in a log message.

```
12/06/19 15:32:19 INFO source.NetcatSource: Source starting
12/06/19 15:32:19 INFO source.NetcatSource: Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
12/06/19 15:32:34 INFO sink.LoggerSink: Event: { headers:{} body: 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 0D          Hello world!. }
```

Congratulations - you've successfully configured and deployed a Flume agent! Subsequent sections cover agent configuration in much more detail.

Data ingestion

Flume supports a number of mechanisms to ingest data from external sources.

RPC

An Avro client included in the Flume distribution can send a given file to Flume Avro source using avro RPC mechanism:

```
$ bin/flume-ng avro-client -H localhost -p 41414 -F /usr/logs/log.10
```

The above command will send the contents of /usr/logs/log.10 to to the Flume source listening on that ports.

Executing commands

There's an exec source that executes a given command and consumes the output. A single 'line' of output ie. text followed by carriage return ('\r') or line feed ('\n') or both together.

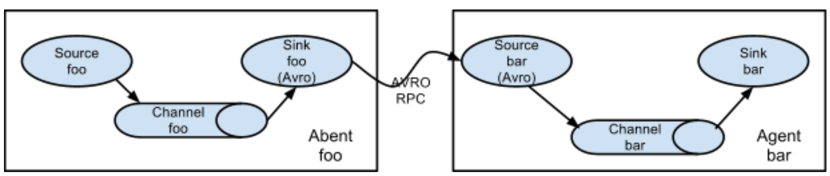
Note: Flume does not support tail as a source. One can wrap the tail command in an exec source to stream the file.

Network streams

Flume supports the following mechanisms to read data from popular log stream types, such as:

1. Avro
2. Syslog
3. Netcat

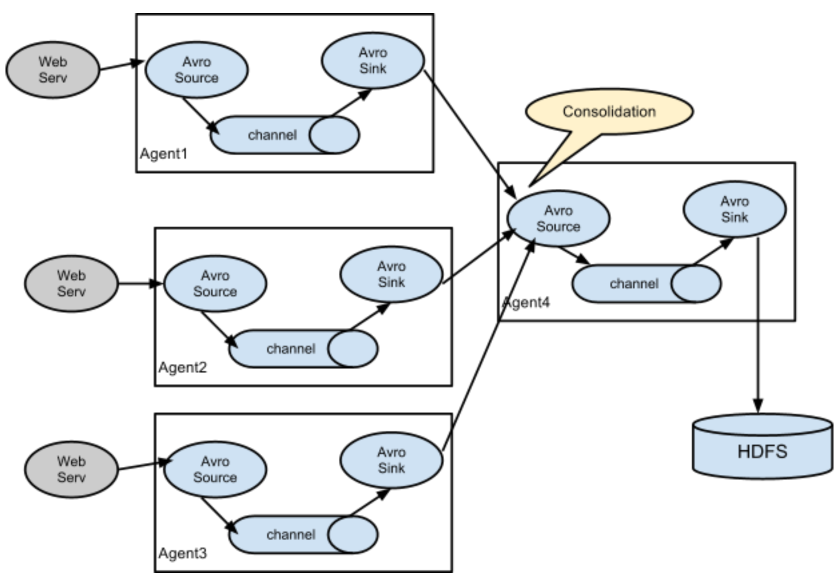
Setting multi-agent flow



In order to flow the data across multiple agents or hops, the sink of the previous agent and source of the current hop need to be avro type with the sink pointing to the hostname (or IP address) and port of the source.

Consolidation

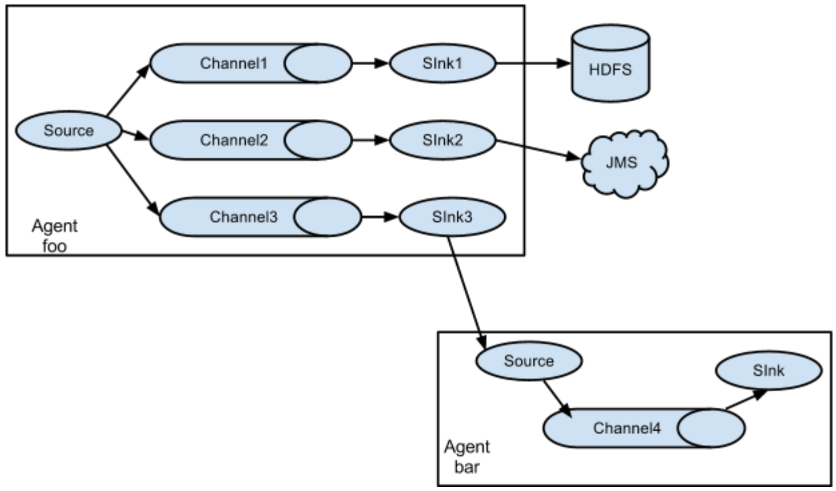
A very common scenario in log collection is a large number of log producing clients sending data to a few consumer agents that are attached to the storage subsystem. For examples, logs collected from hundreds of web servers sent to a dozen of agents that write to HDFS cluster.



This can be achieved in Flume by configuring a number of first tier agents with an avro sink, all pointing to an avro source of single agent. This source on the second tier agent consolidates the received events into a single channel which is consumed by a sink to its final destination.

Multiplexing the flow

Flume supports multiplexing the event flow to one or more destinations. This is achieved by defining a flow multiplexer that can replicate or selectively route an event to one or more channels.



The above example shows a source from agent "foo" fanning out the flow to three different channels. This fan out can be replicating or multiplexing. In case of replicating flow, each event is sent to all three channels. For the multiplexing case, an event is delivered to a subset of available channels when an event's attribute matches a preconfigured value. For example, if an event attribute called "txnType" is set to "customer", then it should go to channel1 and channel3, if it's "vendor" then it should go to channel2, otherwise channel3. The mapping can be set in the agent's configuration file.

Configuration

As mentioned in the earlier section, Flume agent configuration is read from a file that resembles a Java property file format with hierarchical property settings.

Defining the flow

To define the flow within a single agent, you need to link the sources and sinks via a channel. You need to list the sources, sinks and channels for the given agent, and then point the source and sink to a channel. A source instance can specify multiple channels, but a sink instance can only specify on channel. The format is as follows:

```
# list the sources, sinks and channels for the agent
<Agent>.sources = <Source>
<Agent>.sinks = <Sink>
<Agent>.channels = <Channel1> <Channel2>

# set channel for source
<Agent>.sources.<Source>.channels = <Channel1> <Channel2> ...

# set channel for sink
<Agent>.sinks.<Sink>.channel = <Channel1>
```

For example an agent named agent_foo is reading data from an external avro client and sending it to HDFS via a memory channel. The config file weblog.config could look like:

```
# list the sources, sinks and channels for the agent
agent_foo.sources = avro-appserver-src-1
agent_foo.sinks = hdfs-sink-1
agent_foo.channels = mem-channel-1

# set channel for source
agent_foo.sources.avro-appserver-src-1.channels = mem-channel-1

# set channel for sink
agent_foo.sinks.hdfs-sink-1.channel = mem-channel-1
```

This will make the events flow from avro-AppSrv-source to hdfs-Cluster1-sink through the memory channel mem-channel-1. When the agent is started with the weblog.config as its config file, it will instantiate that flow.

Configuring individual components

After defining the flow, you need to set properties of each source, sink and channel. This is done in the same hierarchical namespace fashion where you set the component type and other values for the properties specific to each component:

```
# properties for sources
<Agent>.sources.<Source>.<someProperty> = <someValue>

# properties for channels
<Agent>.channel.<Channel>.<someProperty> = <someValue>

# properties for sinks
<Agent>.sources.<Sink>.<someProperty> = <someValue>
```

The property “type” needs to be set for each component for Flume to understand what kind of object it needs to be. Each source, sink and channel type has its own set of properties required for it to function as intended. All those need to be set as needed. In the previous example, we have a flow from avro-AppSrv-source to hdfs-Cluster1-sink through the memory channel mem-channel-1. Here’s an example that shows configuration of each of those components:

```
agent_foo.sources = avro-AppSrv-source
agent_foo.sinks = hdfs-Cluster1-sink
agent_foo.channels = mem-channel-1

# set channel for sources, sinks

# properties of avro-AppSrv-source
agent_foo.sources.avro-AppSrv-source.type = avro
agent_foo.sources.avro-AppSrv-source.bind = localhost
agent_foo.sources.avro-AppSrv-source.port = 10000

# properties of mem-channel-1
agent_foo.channels.mem-channel-1.type = memory
agent_foo.channels.mem-channel-1.capacity = 1000
agent_foo.channels.mem-channel-1.transactionCapacity = 100

# properties of hdfs-Cluster1-sink
agent_foo.sinks.hdfs-Cluster1-sink.type = hdfs
agent_foo.sinks.hdfs-Cluster1-sink.hdfs.path = hdfs://namenode/flume/webdata

#...
```

Adding multiple flows in an agent

A single Flume agent can contain several independent flows. You can list multiple sources, sinks and channels in a config. These components can be linked to form multiple flows:

```
# list the sources, sinks and channels for the agent
<Agent>.sources = <Source1> <Source2>
<Agent>.sinks = <Sink1> <Sink2>
<Agent>.channels = <Channel1> <Channel2>
```

Then you can link the sources and sinks to their corresponding channels (for sources) of channel (for sinks) to setup two different flows. For example, if you need to setup two flows in an agent, one going from an external avro client to external HDFS and another from output of a tail to avro sink, then here’s a config to do that:

```
# list the sources, sinks and channels in the agent
agent_foo.sources = avro-AppSrv-source1 exec-tail-source2
agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-sink2
agent_foo.channels = mem-channel-1 file-channel-2

# flow #1 configuration
agent_foo.sources.avro-AppSrv-source1.channels = mem-channel-1
agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel-1

# flow #2 configuration
agent_foo.sources.exec-tail-source2.channels = file-channel-2
agent_foo.sinks.avro-forward-sink2.channel = file-channel-2
```

Configuring a multi agent flow

To setup a multi-tier flow, you need to have an avro sink of first hop pointing to avro source of the next hop. This will result in the first Flume agent forwarding events to the next Flume agent. For example, if you are periodically sending files (1 file per event) using avro client to a local Flume agent, then this local agent can forward it to another agent that has the mounted for storage.

Weblog agent config:

```
# list sources, sinks and channels in the agent
agent_foo.sources = avro-AppSrv-source
agent_foo.sinks = avro-forward-sink
agent_foo.channels = file-channel

# define the flow
agent_foo.sources.avro-AppSrv-source.channels = file-channel
agent_foo.sinks.avro-forward-sink.channel = file-channel

# avro sink properties
agent_foo.sources.avro-forward-sink.type = avro
agent_foo.sources.avro-forward-sink.hostname = 10.1.1.100
agent_foo.sources.avro-forward-sink.port = 10000

# configure other pieces
```

```
#...
```

HDFS agent config:

```
# list sources, sinks and channels in the agent
agent_foo.sources = avro-collection-source
agent_foo.sinks = hdfs-sink
agent_foo.channels = mem-channel

# define the flow
agent_foo.sources.avro-collection-source.channels = mem-channel
agent_foo.sinks.hdfs-sink.channel = mem-channel

# avro sink properties
agent_foo.sources.avro-collection-source.type = avro
agent_foo.sources.avro-collection-source.bind = 10.1.1.100
agent_foo.sources.avro-collection-source.port = 10000

# configure other pieces
#...
```

Here we link the avro-forward-sink from the weblog agent to the avro-collection-source of the hdfs agent. This will result in the events coming from the external appserver source eventually getting stored in HDFS.

Fan out flow

As discussed in previous section, Flume support fanning out the flow from one source to multiple channels. There are two modes of fan out, replicating and multiplexing. In the replicating flow the event is sent to all the configured channels. In case of multiplexing, the event is sent to only a subset of qualifying channels. To fan out the flow, one needs to specify a list of channels for a source and the policy for the fanning it out. This is done by adding a channel “selector” that can be replicating or multiplexing. Then further specify the selection rules if it’s a multiplexer. If you don’t specify an selector, then by default it’s replicating:

```
# List the sources, sinks and channels for the agent
<Agent>.sources = <Source1>
<Agent>.sinks = <Sink1> <Sink2>
<Agent>.channels = <Channel1> <Channel2>

# set list of channels for source (separated by space)
<Agent>.sources.<Source1>.channels = <Channel1> <Channel2>

# set channel for sinks
<Agent>.sinks.<Sink1>.channel = <Channel1>
<Agent>.sinks.<Sink2>.channel = <Channel2>

<Agent>.sources.<Source1>.selector.type = replicating
```

The multiplexing select has a further set of properties to bifurcate the flow. This requires specifying a mapping of an event attribute to a set for channel. The selector checks for each configured attribute in the event header. If it matches the specified value, then that event is sent to all the channels mapped to that value. If there’s no match, then the event is sent to set of channels configured as default:

```
# Mapping for multiplexing selector
<Agent>.sources.<Source1>.selector.type = multiplexing
<Agent>.sources.<Source1>.selector.header = <someHeader>
<Agent>.sources.<Source1>.selector.mapping.<Value1> = <Channel1>
<Agent>.sources.<Source1>.selector.mapping.<Value2> = <Channel1> <Channel2>
<Agent>.sources.<Source1>.selector.mapping.<Value3> = <Channel2>
#...

<Agent>.sources.<Source1>.selector.default = <Channel2>
```

The mapping allows overlapping the channels for each value. The default must be set for a multiplexing select which can also contain any number of channels.

The following example has a single flow that multiplexed to two paths. The agent named agent_foo has a single avro source and two channels linked to two sinks:

```
# list the sources, sinks and channels in the agent
agent_foo.sources = avro-AppSrv-source1
agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-sink2
agent_foo.channels = mem-channel-1 file-channel-2

# set channels for source
agent_foo.sources.avro-AppSrv-source1.channels = mem-channel-1 file-channel-2

# set channel for sinks
agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel-1
agent_foo.sinks.avro-forward-sink2.channel = file-channel-2

# channel selector configuration
agent_foo.sources.avro-AppSrv-source1.selector.type = multiplexing
agent_foo.sources.avro-AppSrv-source1.selector.header = State
agent_foo.sources.avro-AppSrv-source1.selector.mapping.CA = mem-channel-1
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.mapping.NY = mem-channel-1 file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.default = mem-channel-1
```

The selector checks for a header called “State”. If the value is “CA” then its sent to mem-channel-1, if its “AZ” then it goes to file-channel-2 or if its “NY” then both. If the “State” header is not set or doesn’t match any of the three, then it goes to mem-channel-1 which is designated as ‘default’.

The selector also supports optional channels. To specify optional channels for a header, the config parameter ‘optional’ is used in the following way:

```
# channel selector configuration
agent_foo.sources.avro-AppSrv-source1.selector.type = multiplexing
agent_foo.sources.avro-AppSrv-source1.selector.header = State
agent_foo.sources.avro-AppSrv-source1.selector.mapping.CA = mem-channel-1
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.mapping.NY = mem-channel-1 file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.optional.CA = mem-channel-1 file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.default = mem-channel-1
```

The selector will attempt to write to the required channels first and will fail the transaction if even one of these channels fails to consume the events. The transaction is reattempted on **all** of the channels. Once all required channels have consumed the events, then the selector will attempt to write to the optional channels. A failure by any of the optional channels to consume the event is simply ignored and not retried.

If there is an overlap between the optional channels and required channels for a specific header, the channel is considered to be required, and a failure in the channel will cause the entire set of required channels to be retried. For instance, in the above example, for the header “CA” mem-channel-1 is considered to be a required channel even though it is marked both as required and optional, and a failure to write to this channel will cause that event to be retried on **all** channels configured for the selector.

Note that if a header does not have any required channels, then the event will be written to the default channels and will be attempted to be written to the optional channels for that header. Specifying optional channels will still cause the event to be written to the default channels, if no required channels are specified.

Flume Sources

Avro Source

Listens on Avro port and receives events from external Avro client streams. When paired with the built-in AvroSink on another (previous hop) Flume agent, it can create tiered collection topologies. Required properties are in **bold**.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `avro`
bind	–	hostname or IP address to listen on
port	–	Port # to bind to
threads	–	Maximum number of worker threads to spawn
selector.type		
selector.*		
interceptors	–	Space separated list of interceptors
interceptors.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = avro
a1.sources.r1.channels = c1
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.port = 4141
```

Exec Source

Exec source runs a given Unix command on start-up and expects that process to continuously produce data on standard out (stderr is simply discarded, unless property logStdErr is set to true). If the process exits for any reason, the source also exits and will produce no further data. This means configurations such as `cat [named pipe]` or `tail -F [file]` are going to produce the desired results where as `date` will probably not - the former two commands produce streams of data where as the latter produces a single event and exits.

Required properties are in **bold**.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `exec`
command	–	The command to execute
restartThrottle	10000	Amount of time (in millis) to wait before attempting a restart
restart	false	Whether the executed cmd should be restarted if it dies
logStdErr	false	Whether the command’s stderr should be logged
batchSize	20	The max number of lines to read and send to the channel at a time
selector.type	replicating	replicating or multiplexing
selector.*		Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

Warning: The problem with ExecSource and other asynchronous sources is that the source can not guarantee that if there is a failure to put the event into the Channel the client knows about it. In such cases, the data will be lost. As a for instance, one of the most commonly requested features is the `tail -F [file]`-like use case where an application writes to a log file on disk and Flume tails the file, sending each line as an event. While this is possible, there's an obvious problem; what happens if the channel fills up and Flume can't send an event? Flume has no way of indicating to the application writing the log file that it needs to retain the log or that the event hasn't been sent, for some reason. If this doesn't make sense, you need only know this: Your application can never guarantee data has been received when using a unidirectional asynchronous interface such as ExecSource! As an extension of this warning - and to be completely clear - there is absolutely zero guarantee of event delivery when using this source. For stronger reliability guarantees, consider the Spooling Directory Source or direct integration with Flume via the SDK.

Note: You can use ExecSource to emulate TailSource from Flume 0.9x (flume og). Just use unix command `tail -F /full/path/to/your/file`. Parameter -F is better in this case than -f as it will also follow file rotation.

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F /var/log/secure
a1.sources.r1.channels = c1
```

Spooling Directory Source

This source lets you ingest data by dropping files in a spooling directory on disk. **Unlike other asynchronous sources, this source avoids data loss even if Flume is restarted or fails.** Flume will watch the directory for new files and read then ingest them as they appear. After a given file has been fully read into the channel, it is renamed to indicate completion. This allows a cleaner process to remove completed files periodically. Note, however, that events may be duplicated if failures occur, consistent with the semantics offered by other Flume components. The channel optionally inserts the full path of the origin file into a header field of each event. This source buffers file data in memory during reads; be sure to set the *bufferMaxLineLength* option to a number greater than the longest line you expect to see in your input data.

Warning: This channel expects that only immutable, uniquely named files are dropped in the spooling directory. If duplicate names are used, or files are modified while being read, the source will fail with an error message. For some use cases this may require adding unique identifiers (such as a timestamp) to log file names when they are copied into the spooling directory.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `spooldir`
spoolDir	–	The directory where log files will be spooled
fileSuffix	.COMPLETED	Suffix to append to completely ingested files
fileHeader	false	Whether to add a header storing the filename
fileHeaderKey	file	Header key to use when appending filename to header
batchSize	10	Granularity at which to batch transfer to the channel
bufferMaxLines	100	Maximum number of lines the commit buffer can hold
bufferMaxLineLength	5000	Maximum length of a line in the commit buffer
selector.type	replicating	replicating or multiplexing
selector.*		Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /var/log/apache/flumeSpool
a1.sources.r1.fileHeader = true
a1.sources.r1.channels = c1
```

NetCat Source

A netcat-like source that listens on a given port and turns each line of text into an event. Acts like `nc -k -l [host] [port]`. In other words, it opens a specified port and listens for data. The expectation is that the supplied data is newline separated text. Each line of text is turned into a Flume event and sent via the connected channel.

Required properties are in **bold**.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `netcat`
bind	–	Host name or IP address to bind to
port	–	Port # to bind to
max-line-length	512	Max line length per event body (in bytes)

ack-every-event	true	Respond with an "OK" for every event received
selector.type	replicating	replicating or multiplexing
selector.*		Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = netcat
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.bind = 6666
a1.sources.r1.channels = c1
```

Sequence Generator Source

A simple sequence generator that continuously generates events with a counter that starts from 0 and increments by 1. Useful mainly for testing. Required properties are in **bold**.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `seq`
selector.type		replicating or multiplexing
selector.*	replicating	Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		
batchSize	1	

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = seq
a1.sources.r1.channels = c1
```

Syslog Sources

Reads syslog data and generate Flume events. The UDP source treats an entire message as a single event. The TCP sources create a new event for each string of characters separated by a newline ('n').

Required properties are in **bold**.

Syslog TCP Source

The original, tried-and-true syslog TCP source.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `syslogtcp`
host	–	Host name or IP address to bind to
port	–	Port # to bind to
eventSize	2500	Maximum size of a single event line, in bytes
selector.type		replicating or multiplexing
selector.*	replicating	Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

For example, a syslog TCP source for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = syslogtcp
a1.sources.r1.port = 5140
a1.sources.r1.host = localhost
a1.sources.r1.channels = c1
```

Multiport Syslog TCP Source

This is a newer, faster, multi-port capable version of the Syslog TCP source. Note that the `ports` configuration setting has replaced `port`. Multi-port capability means that it can listen on many ports at once in an efficient manner. This source uses the Apache Mina library to do that. Provides support for RFC-3164 and many common RFC-5424 formatted messages. Also provides the capability to configure the character set used on a per-port basis.

Property Name	Default	Description

channels	–	
type	–	The component type name, needs to be `multiport_syslogtcp`
host	–	Host name or IP address to bind to.
ports	–	Space-separated list (one or more) of ports to bind to.
eventSize	2500	Maximum size of a single event line, in bytes.
portHeader	–	If specified, the port number will be stored in the header of each event using the header name specified here. This allows for interceptors and channel selectors to customize routing logic based on the incoming port.
charset.default	UTF-8	Default character set used while parsing syslog events into strings.
charset.port.<port>	–	Character set is configurable on a per-port basis.
batchSize	100	Maximum number of events to attempt to process per request loop. Using the default is usually fine.
readBufferSize	1024	Size of the internal Mina read buffer. Provided for performance tuning. Using the default is usually fine.
numProcessors	(auto-detected)	Number of processors available on the system for use while processing messages. Default is to auto-detect # of CPUs using the Java Runtime API. Mina will spawn 2 request-processing threads per detected CPU, which is often reasonable.
selector.type	replicating	replicating, multiplexing, or custom
selector.*	–	Depends on the `selector.type` value
interceptors	–	Space separated list of interceptors.
interceptors.*		

For example, a multiport syslog TCP source for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = multiport_syslogtcp
a1.sources.r1.channels = c1
a1.sources.r1.host = 0.0.0.0
a1.sources.r1.ports = 10001 10002 10003
a1.sources.r1.portHeader = port
```

Syslog UDP Source

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `syslogudp`
host	–	Host name or IP address to bind to
port	–	Port # to bind to
selector.type		replicating or multiplexing
selector.*	replicating	Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

For example, a syslog UDP source for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = syslogudp
a1.sources.r1.port = 5140
a1.sources.r1.host = localhost
a1.sources.r1.channels = c1
```

HTTP Source

A source which accepts Flume Events by HTTP POST and GET. GET should be used for experimentation only. HTTP requests are converted into flume events by a pluggable "handler" which must implement the HTTPSourceHandler interface. This handler takes a HttpServletRequest and returns a list of flume events. All events handler from one Http request are committed to the channel in one transaction, thus allowing for increased efficiency on channels like the file channel. If the handler throws an exception this source will return a HTTP status of 400. If the channel is full, or the source is unable to append events to the channel, the source will return a HTTP 503 - Temporarily unavailable status.

All events sent in one post request are considered to be one batch and inserted into the channel in one transaction.

Property Name	Default	Description
type		The FQCN of this class: `org.apache.flume.source.http.HTTPSource`
port	–	The port the source should bind to.
handler	`org.apache.flume.http.JSONHandler`	The FQCN of the handler class.
handler.*	–	Config parameters for the handler
selector.type	replicating	replicating or multiplexing
selector.*		Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

For example, a http source for agent named a1:

```
a1.sources = r1
a1.channels = c1
```

```
a1.sources.r1.type = org.apache.flume.source.http.HTTPSource
a1.sources.r1.port = 5140
a1.sources.r1.channels = c1
a1.sources.r1.handler = org.example.rest.RestHandler
a1.sources.r1.handler.nickname = random props
```

JSONHandler

A handler is provided out of the box which can handle events represented in JSON format, and supports UTF-8, UTF-16 and UTF-32 character sets. The handler accepts an array of events (even if there is only one event, the event has to be sent in an array) and converts them to a Flume event based on the encoding specified in the request. If no encoding is specified, UTF-8 is assumed. The JSON handler supports UTF-8, UTF-16 and UTF-32. Events are represented as follows.

```
[{
  "headers" : {
             "timestamp" : "434324343",
             "host" : "random_host.example.com"
             },
  "body" : "random_body"
  },
  {
  "headers" : {
             "namenode" : "namenode.example.com",
             "datanode" : "random_datanode.example.com"
             },
  "body" : "really_random_body"
  }]
```

To set the charset, the request must have content type specified as `application/json; charset=UTF-8` (replace UTF-8 with UTF-16 or UTF-32 as required).

One way to create an event in the format expected by this handler, is to use JSONEvent provided in the Flume SDK and use Google Gson to create the JSON string using the Gson#fromJson(Object, Type) method. The type token to pass as the 2nd argument of this method for list of events can be created by:

```
Type type = new TypeToken<List<JSONEvent>>() {}.getType();
```

Legacy Sources

The legacy sources allow a Flume 1.x agent to receive events from Flume 0.9.4 agents. It accepts events in the Flume 0.9.4 format, converts them to the Flume 1.0 format, and stores them in the connected channel. The 0.9.4 event properties like timestamp, pri, host, nanos, etc get converted to 1.x event header attributes. The legacy source supports both Avro and Thrift RPC connections. To use this bridge between two Flume versions, you need to start a Flume 1.x agent with the avroLegacy or thriftLegacy source. The 0.9.4 agent should have the agent Sink pointing to the host/port of the 1.x agent.

> **Note:** The reliability semantics of Flume 1.x are different from that of Flume 0.9.x. The E2E or DFO mode of a Flume 0.9.x agent will not be supported by the legacy source. The only supported 0.9.x mode is the best effort, though the reliability setting of the 1.x flow will be applicable to the events once they are saved into the Flume 1.x channel by the legacy source.

Required properties are in **bold**.

Avro Legacy Source

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `org.apache.flume.source.avroLegacy.AvroLegacySource`
host	–	The hostname or IP address to bind to
port	–	The port # to listen on
selector.type		replicating or multiplexing
selector.*	replicating	Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.avroLegacy.AvroLegacySource
a1.sources.r1.host = 0.0.0.0
a1.sources.r1.bind = 6666
a1.sources.r1.channels = c1
```

Thrift Legacy Source

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be `org.apache.flume.source.thriftLegacy.ThriftLegacySource`
host	–	The hostname or IP address to bind to
port	–	The port # to listen on
selector.type		replicating or multiplexing

selector.type		replicating or multiplexing
selector.*	replicating	Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.thriftLegacy.ThriftLegacySource
a1.sources.r1.host = 0.0.0.0
a1.sources.r1.bind = 6666
a1.sources.r1.channels = c1
```

Custom Source

A custom source is your own implementation of the Source interface. A custom source's class and its dependencies must be included in the agent's classpath when starting the Flume agent. The type of the custom source is its FQCN.

Property Name	Default	Description
channels	–	
type	–	The component type name, needs to be your FQCN
selector.type		`replicating` or `multiplexing`
selector.*	replicating	Depends on the selector.type value
interceptors	–	Space separated list of interceptors
interceptors.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.example.MySource
a1.sources.r1.channels = c1
```

Scribe Source

Scribe is another type of ingest system. To adopt existing Scribe ingest system, Flume should use ScribeSource based on Thrift with compatible transfering protocol. The deployment of Scribe please following guide from Facebook. Required properties are in **bold**.

Property Name	Default	Description
type	–	The component type name, needs to be `org.apache.flume.source.scribe.ScribeSource`
port	1499	Port that Scribe should be connected
workerThreads	5	Handing threads number in Thrift
selector.type		
selector.*		

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.scribe.ScribeSource
a1.sources.r1.port = 1463
a1.sources.r1.workerThreads = 5
a1.sources.r1.channels = c1
```

Flume Sinks

HDFS Sink

This sink writes events into the Hadoop Distributed File System (HDFS). It currently supports creating text and sequence files. It supports compression in both file types. The files can be rolled (close current file and create a new one) periodically based on the elapsed time or size of data or number of events. It also buckets/partitions data by attributes like timestamp or machine where the event originated. The HDFS directory path may contain formatting escape sequences that will replaced by the HDFS sink to generate a directory/file name to store the events. Using this sink requires hadoop to be installed so that Flume can use the Hadoop jars to communicate with the HDFS cluster. Note that a version of Hadoop that supports the sync() call is required.

The following are the escape sequences supported:

Alias	Description
%{host}	Substitute value of event header named "host". Arbitrary header names are supported.
%t	Unix time in milliseconds
%a	locale's short weekday name (Mon, Tue, ...)
%A	locale's full weekday name (Monday, Tuesday, ...)
%b	locale's short month name (Jan, Feb, ...)

%B	locale's long month name (January, February, ...)
%c	locale's date and time (Thu Mar 3 23:05:25 2005)
%d	day of month (01)
%D	date; same as %m/%d/%y
%H	hour (00..23)
%I	hour (01..12)
%j	day of year (001..366)
%k	hour (0..23)
%m	month (01..12)
%M	minute (00..59)
%p	locale's equivalent of am or pm
%s	seconds since 1970-01-01 00:00:00 UTC
%S	second (00..59)
%y	last two digits of year (00..99)
%Y	year (2010)
%z	+hhmm numeric timezone (for example, -0400)

The file in use will have the name mangled to include ".tmp" at the end. Once the file is closed, this extension is removed. This allows excluding partially complete files in the directory. Required properties are in **bold**.

> **Note:** For all of the time related escape sequences, a header with the key "timestamp" must exist among the headers of the event. One way to add this automatically is to use the TimestampInterceptor.

Name	Default	Description
channel	–	
type	–	The component type name, needs to be `hdfs`
hdfs.path	–	HDFS directory path (eg hdfs://namenode/flume/webdata/)
hdfs.filePrefix	FlumeData	Name prefixed to files created by Flume in hdfs directory
hdfs.fileSuffix	–	Suffix to append to file (eg `.avro` - *NOTE: period is not automatically added*)
hdfs.rollInterval	30	Number of seconds to wait before rolling current file (0 = never roll based on time interval)
hdfs.rollSize	1024	File size to trigger roll, in bytes (0: never roll based on file size)
hdfs.rollCount	10	Number of events written to file before it rolled (0 = never roll based on number of events)
hdfs.idleTimeout	0	Timeout after which inactive files get closed (0 = disable automatic closing of idle files)
hdfs.batchSize	100	number of events written to file before it is flushed to HDFS
hdfs.codeC	–	Compression codec. one of following : gzip, bzip2, lzo, snappy
hdfs.fileType	SequenceFile	File format: currently `SequenceFile`, `DataStream` or `CompressedStream` (1)DataStream will not compress output file and please don't set codeC (2)CompressedStream requires set hdfs.codeC with an available codeC
hdfs.maxOpenFiles	5000	Allow only this number of open files. If this number is exceeded, the oldest file is closed.
hdfs.writeFormat	–	"Text" or "Writable"
hdfs.callTimeout	10000	Number of milliseconds allowed for HDFS operations, such as open, write, flush, close. This number should be increased if many HDFS timeout operations are occurring.
hdfs.threadsPoolSize	10	Number of threads per HDFS sink for HDFS IO ops (open, write, etc.)
hdfs.rollTimerPoolSize	1	Number of threads per HDFS sink for scheduling timed file rolling
hdfs.kerberosPrincipal	–	Kerberos user principal for accessing secure HDFS
hdfs.kerberosKeytab	–	Kerberos keytab for accessing secure HDFS
hdfs.proxyUser		
hdfs.round	false	Should the timestamp be rounded down (if true, affects all time based escape sequences except %t)
hdfs.roundValue	1	Rounded down to the highest multiple of this (in the unit configured using `hdfs.roundUnit`), less than current time.
hdfs.roundUnit	second	The unit of the round down value - `second`, `minute` or `hour`.
hdfs.timeZone	Local Time	Name of the timezone that should be used for resolving the directory path, e.g. America/Los_Angeles.
serializer	`TEXT`	Other possible options include `avro_event` or the fully-qualified class name of an implementation of the `EventSerializer.Builder` interface.
serializer.*		

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
a1.sinks.k1.hdfs.filePrefix = events-
a1.sinks.k1.hdfs.round = true
a1.sinks.k1.hdfs.roundValue = 10
a1.sinks.k1.hdfs.roundUnit = minute
```

The above configuration will round down the timestamp to the last 10th minute. For example, an event with timestamp 11:54:34 AM, June 12, 2012 will cause the hdfs path to become `/flume/events/2012-06-12/1150/00`.

Logger Sink

Logs event at INFO level. Typically useful for testing/debugging purpose. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `logger`

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = logger
a1.sinks.k1.channel = c1
```

Avro Sink

This sink forms one half of Flume's tiered collection support. Flume events sent to this sink are turned into Avro events and sent to the configured hostname / port pair. The events are taken from the configured Channel in batches of the configured batch size. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `avro`.
hostname	–	The hostname or IP address to bind to.
port	–	The port # to listen on.
batch-size	100	number of event to batch together for send.
connect-timeout	20000	Amount of time (ms) to allow for the first (handshake) request.
request-timeout	20000	Amount of time (ms) to allow for requests after the first.

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = avro
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = 10.10.10.10
a1.sinks.k1.port = 4545
```

IRC Sink

The IRC sink takes messages from attached channel and relays those to configured IRC destinations. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `irc`
hostname	–	The hostname or IP address to connect to
port	6667	The port number of remote host to connect
nick	–	Nick name
user	–	User name
password	–	User password
chan	–	channel
name		
splitlines	–	(boolean)
splitchars	n	line separator (if you were to enter the default value into the config file, then you would need to escape the backslash, like this: "\n")

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = irc
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = irc.yourdomain.com
a1.sinks.k1.nick = flume
a1.sinks.k1.chan = #flume
```

File Roll Sink

Stores events on the local filesystem. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `file_roll`.
sink.directory	–	The directory where files will be stored
sink.rollInterval	30	Roll the file every 30 seconds. Specifying 0 will disable rolling and cause all events to be written to a single file.
sink.serializer	TEXT	Other possible options include `avro_event` or the FQCN of an implementation of EventSerializer.Builder interface.
batchSize	100	

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = file_roll
a1.sinks.k1.channel = c1
a1.sinks.k1.sink.directory = /var/log/flume
```

Null Sink

Discards all events it receives from the channel. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `null`.
batchSize	100	

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = null
a1.sinks.k1.channel = c1
```

HBaseSinks

HBaseSink

This sink writes data to HBase. The Hbase configuration is picked up from the first hbase-site.xml encountered in the classpath. A class implementing HbaseEventSerializer which is specified by the configuration is used to convert the events into HBase puts and/or increments. These puts and increments are then written to HBase. This sink provides the same consistency guarantees as HBase, which is currently row-wise atomicity. In the event of Hbase failing to write certain events, the sink will replay all events in that transaction. For convenience two serializers are provided with flume. The SimpleHbaseEventSerializer (org.apache.flume.sink.hbase.SimpleHbaseEventSerializer) writes the event body as is to HBase, and optionally increments a column in Hbase. This is primarily an example implementation. The RegexHbaseEventSerializer (org.apache.flume.sink.hbase.RegexHbaseEventSerializer) breaks the event body based on the given regex and writes each part into different columns.

The type is the FQCN: org.apache.flume.sink.hbase.HBaseSink. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `org.apache.flume.sink.hbase.HBaseSink`
table	–	The name of the table in Hbase to write to.
columnFamily	–	The column family in Hbase to write to.
batchSize	100	Number of events to be written per txn.
serializer	org.apache.flume.sink.hbase.SimpleHbaseEventSerializer	
serializer.*	–	Properties to be passed to the serializer.

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = org.apache.flume.sink.hbase.HBaseSink
a1.sinks.k1.table = foo_table
a1.sinks.k1.columnFamily = bar_cf
a1.sinks.k1.serializer = org.apache.flume.sink.hbase.RegexHbaseEventSerializer
a1.sinks.k1.channel = c1
```

AsyncHBaseSink

This sink writes data to HBase using an asynchronous model. A class implementing AsyncHbaseEventSerializer which is specified by the configuration is used to convert the events into HBase puts and/or increments. These puts and increments are then written to HBase. This sink provides the same consistency guarantees as HBase, which is currently row-wise atomicity. In the event of Hbase failing to write certain events, the sink will replay all events in that transaction. This sink is still experimental. The type is the FQCN: org.apache.flume.sink.hbase.AsyncHBaseSink. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `org.apache.flume.sink.hbase.AsyncHBaseSink`
table	–	The name of the table in Hbase to write to.
columnFamily	–	The column family in Hbase to write to.
batchSize	100	Number of events to be written per txn.
timeout	–	The length of time (in milliseconds) the sink waits for acks from hbase for all events in a transaction. If no timeout is specified, the sink will wait forever.

serializer	org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer	
serializer.*	–	Properties to be passed to the serializer.

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = org.apache.flume.sink.hbase.AsyncHBaseSink
a1.sinks.k1.table = foo_table
a1.sinks.k1.columnFamily = bar_cf
a1.sinks.k1.serializer = org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer
a1.sinks.k1.channel = c1
```

ElasticSearchSink

This sink writes data to ElasticSearch. A class implementing ElasticSearchEventSerializer which is specified by the configuration is used to convert the events into XContentBuilder which detail the fields and mappings which will be indexed. These are then then written to ElasticSearch. The sink will generate an index per day allowing easier management instead of dealing with a single large index The type is the FQCN: org.apache.flume.sink.elasticsearch.ElasticSearchSink Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be `org.apache.flume.sink.elasticsearch.ElasticSearchSink`
hostNames	–	Comma separated list of hostname:port, if the port is not present the default port '9300' will be used
indexName	flume	The name of the index which the date will be appended to. Example 'flume' -> 'flume-yyyy-MM-dd'
indexType	logs	The type to index the document to, defaults to 'log'
clusterName	elasticsearch	Name of the ElasticSearch cluster to connect to
batchSize	100	Number of events to be written per txn.
ttl	–	TTL in days, when set will cause the expired documents to be deleted automatically, if not set documents will never be automatically deleted
serializer	org.apache.flume.sink.elasticsearch.ElasticSearchDynamicSerializer	
serializer.*	–	Properties to be passed to the serializer.

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = org.apache.flume.sink.elasticsearch.ElasticSearchSink
a1.sinks.k1.hostNames = 127.0.0.1:9200,127.0.0.2:9300
a1.sinks.k1.indexName = foo_index
a1.sinks.k1.indexType = bar_type
a1.sinks.k1.clusterName = foobar_cluster
a1.sinks.k1.batchSize = 500
a1.sinks.k1.ttl = 5
a1.sinks.k1.serializer = org.apache.flume.sink.elasticsearch.ElasticSearchDynamicSerializer
a1.sinks.k1.channel = c1
```

Custom Sink

A custom sink is your own implementation of the Sink interface. A custom sink's class and its dependencies must be included in the agent's classpath when starting the Flume agent. The type of the custom sink is its FQCN. Required properties are in **bold**.

Property Name	Default	Description
channel	–	
type	–	The component type name, needs to be your FQCN

Example for agent named a1:

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = org.example.MySink
a1.sinks.k1.channel = c1
```

Flume Channels

Channels are the repositories where the events are staged on a agent. Source adds the events and Sink removes it.

Memory Channel

The events are stored in a an in-memory queue with configurable max size. It's ideal for flow that needs higher throughput and prepared to lose the staged data in the event of a agent failures. Required properties are in **bold**.

Property Name	Default	Description
type	–	The component type name, needs to be `memory`
capacity	100	The max number of events stored in the channel
transactionCapacity	100	The max number of events stored in the channel per transaction
keep-alive	3	Timeout in seconds for adding or removing an event

Example for agent named a1:

```
a1.channels = c1
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
```

JDBC Channel

The events are stored in a persistent storage that's backed by a database. The JDBC channel currently supports embedded Derby. This is a durable channel that's ideal for the flows where recoverability is important. Required properties are in **bold**.

Property Name	Default	Description
type	–	The component type name, needs to be `jdbc`
db.type	DERBY	Database vendor, needs to be DERBY.
driver.class	org.apache.derby.jdbc.EmbeddedDriver	Class for vendor's JDBC driver
driver.url	(constructed from other properties)	JDBC connection URL
db.username	"sa"	User id for db connection
db.password	–	password for db connection
connection.properties.file	–	JDBC Connection property file path
create.schema	true	If true, then creates db schema if not there
create.index	true	Create indexes to speed up lookups
create.foreignkey	true	
transaction.isolation	"READ_COMMITTED"	Isolation level for db session READ_UNCOMMITTED, READ_COMMITTED, SERIALIZABLE, REPEATABLE_READ
maximum.connections	10	Max connections allowed to db
maximum.capacity	0 (unlimited)	Max number of events in the channel
sysprop.*		DB Vendor specific properties
sysprop.user.home		Home path to store embedded Derby database

Example for agent named a1:

```
a1.channels = c1
a1.channels.c1.type = jdbc
```

Recoverable Memory Channel

> **Warning:** The Recoverable Memory Channel has been deprecated in favor of the FileChannel. FileChannel is durable channel and performs better than the Recoverable Memory Channel.

Required properties are in **bold**.

Property Name	Default	Description
type	–	The component type name, needs to be `org.apache.flume.channel.recoverable.memory.RecoverableMemoryChannel`
wal.dataDir	${user.home}/.flume/recoverable-memory-channel	
wal.rollSize	(0x04000000)	Max size (in bytes) of a single file before we roll
wal.minRetentionPeriod	300000	Min amount of time (in millis) to keep a log
wal.workerInterval	60000	How often (in millis) the background worker checks for old logs
wal.maxLogsSize	(0x20000000)	Total amt (in bytes) of logs to keep, excluding the current log
capacity	100	
transactionCapacity	100	
keep-alive	3	

File Channel

Required properties are in **bold**.

Property Name Default	Description	
type	–	The component type name, needs to be `file`.
checkpointDir	~/.flume/file-channel/checkpoint	The directory where checkpoint file will be stored
dataDirs	~/.flume/file-channel/data	The directory where log files will be stored
transactionCapacity	1000	The maximum size of transaction supported by the channel
checkpointInterval	30000	Amount of time (in millis) between checkpoints
maxFileSize	2146435071	Max size (in bytes) of a single log file
minimumRequiredSpace	524288000	Minimum Required free space (in bytes)

capacity	1000000	Maximum capacity of the channel
keep-alive	3	Amount of time (in sec) to wait for a put operation
write-timeout	3	Amount of time (in sec) to wait for a write operation
checkpoint-timeout	600	Expert: Amount of time (in sec) to wait for a checkpoint
use-log-replay-v1	false	Expert: Use old replay logic
use-fast-replay	false	Expert: Replay without using queue
encryption.activeKey	–	Key name used to encrypt new data
encryption.cipherProvider	–	Cipher provider type, supported types: AESCTRNOPADDING
encryption.keyProvider	–	Key provider type, supported types: JCEKSFILE
encryption.keyProvider.keyStoreFile	–	Path to the keystore file
encrpytion.keyProvider.keyStorePasswordFile	–	Path to the keystore password file
encryption.keyProvider.keys	–	List of all keys (e.g. history of the activeKey setting)
encyption.keyProvider.keys.*.passwordFile	–	Path to the optional key password file

Note: By default the File Channel uses paths for checkpoint and data directories that are within the user home as specified above. As a result if you have more than one File Channel instances active within the agent, only one will be able to lock the directories and cause the other channel initialization to fail. It is therefore necessary that you provide explicit paths to all the configured channels, preferably on different disks. Furthermore, as file channel will sync to disk after every commit, coupling it with a sink/source that batches events together may be necessary to provide good performance where multiple disks are not available for checkpoint and data directories.

Example for agent named a1:

```
a1.channels = c1
a1.channels.c1.type = file
a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
a1.channels.c1.dataDirs = /mnt/flume/data
```

Encryption

Below is a few sample configurations:

Generating a key with a password seperate from the key store password:

```
keytool -genseckey -alias key-0 -keypass keyPassword -keyalg AES \
  -keysize 128 -validity 9000 -keystore test.keystore \
  -storetype jceks -storepass keyStorePassword
```

Generating a key with the password the same as the key store password:

```
keytool -genseckey -alias key-1 -keyalg AES -keysize 128 -validity 9000 \
  -keystore src/test/resources/test.keystore -storetype jceks \
  -storepass keyStorePassword
```

```
a1.channels.c1.encryption.activeKey = key-0
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.keyProvider = key-provider-0
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile = /path/to/my.keystore.password
a1.channels.c1.encryption.keyProvider.keys = key-0
```

Let's say you have aged key-0 out and new files should be encrypted with key-1:

```
a1.channels.c1.encryption.activeKey = key-1
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile = /path/to/my.keystore.password
a1.channels.c1.encryption.keyProvider.keys = key-0 key-1
```

The same scenerio as above, however key-0 has it's own password:

```
a1.channels.c1.encryption.activeKey = key-1
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile = /path/to/my.keystore.password
a1.channels.c1.encryption.keyProvider.keys = key-0 key-1
a1.channels.c1.encryption.keyProvider.keys.key-0.passwordFile = /path/to/key-0.password
```

Pseudo Transaction Channel

Warning: The Pseudo Transaction Channel is only for unit testing purposes and is NOT meant for production use.

Required properties are in **bold**.

Property Name	Default	Description
type	–	The component type name, needs to be `org.apache.flume.channel.PseudoTxnMemoryChannel`

capacity	50	The max number of events stored in the channel
keep-alive	3	Timeout in seconds for adding or removing an event

Custom Channel

A custom channel is your own implementation of the Channel interface. A custom channel's class and its dependencies must be included in the agent's classpath when starting the Flume agent. The type of the custom channel is its FQCN. Required properties are in **bold**.

Property Name	Default	Description
type	–	The component type name, needs to be a FQCN

Example for agent named a1:

```
a1.channels = c1
a1.channels.c1.type = org.example.MyChannel
```

Flume Channel Selectors

If the type is not specified, then defaults to "replicating".

Replicating Channel Selector (default)

Required properties are in **bold**.

Property Name	Default	Description
selector.type	replicating	The component type name, needs to be `replicating`

Example for agent named a1 and it's source called r1:

```
a1.sources = r1
a1.channels = c1 c2 c3
a1.source.r1.selector.type = replicating
a1.source.r1.channels = c1 c2 c3
```

Multiplexing Channel Selector

Required properties are in **bold**.

Property Name	Default	Description
selector.type	replicating	The component type name, needs to be `multiplexing`
selector.header	flume.selector.header	
selector.default	–	
selector.mapping.*	–	

Example for agent named a1 and it's source called r1:

```
a1.sources = r1
a1.channels = c1 c2 c3 c4
a1.sources.r1.selector.type = multiplexing
a1.sources.r1.selector.header = state
a1.sources.r1.selector.mapping.CZ = c1
a1.sources.r1.selector.mapping.US = c2 c3
a1.sources.r1.selector.default = c4
```

Custom Channel Selector

A custom channel selector is your own implementation of the ChannelSelector interface. A custom channel selector's class and its dependencies must be included in the agent's classpath when starting the Flume agent. The type of the custom channel selector is its FQCN.

Property Name	Default	Description
selector.type	–	The component type name, needs to be your FQCN

Example for agent named a1 and it's source called r1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.selector.type = org.example.MyChannelSelector
```

Flume Sink Processors

Sink groups allow users to group multiple sinks into one entity. Sink processors can be used to provide load balancing capabilities over all sinks inside the group or to achieve fail over from one sink to another in case of temporal failure.

Required properties are in **bold**.

Property Name	Default	Description
sinks	–	Space separated list of sinks that are participating in the group
processor.type	`default`	The component type name, needs to be `default`, `failover` or `load_balance`

Example for agent named a1:

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = load_balance
```

Default Sink Processor

Default sink processor accepts only a single sink. User is not forced to create processor (sink group) for single sinks. Instead user can follow the source - channel - sink pattern that was explained above in this user guide.

Failover Sink Processor

Failover Sink Processor maintains a prioritized list of sinks, guaranteeing that so long as one is available events will be processed (delivered).

The fail over mechanism works by relegating failed sinks to a pool where they are assigned a cool down period, increasing with sequential failures before they are retried. Once a sink successfully sends an event it is restored to the live pool.

To configure, set a sink groups processor to `failover` and set priorities for all individual sinks. All specified priorities must be unique. Furthermore, upper limit to fail over time can be set (in milliseconds) using `maxpenalty` property.

Required properties are in **bold**.

Property Name	Default	Description
sinks	–	Space separated list of sinks that are participating in the group
processor.type	`default`	The component type name, needs to be `failover`
processor.priority.<sinkName>	–	<sinkName> must be one of the sink instances associated with the current sink group
processor.maxpenalty	30000	(in millis)

Example for agent named a1:

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = failover
a1.sinkgroups.g1.processor.priority.k1 = 5
a1.sinkgroups.g1.processor.priority.k2 = 10
a1.sinkgroups.g1.processor.maxpenalty = 10000
```

Load balancing Sink Processor

Load balancing sink processor provides the ability to load-balance flow over multiple sinks. It maintains an indexed list of active sinks on which the load must be distributed. Implementation supports distributing load using either via `round_robin` or `random` selection mechanisms. The choice of selection mechanism defaults to `round_robin` type, but can be overridden via configuration. Custom selection mechanisms are supported via custom classes that inherits from `AbstractSinkSelector`.

When invoked, this selector picks the next sink using its configured selection mechanism and invokes it. For `round_robin` and `random` In case the selected sink fails to deliver the event, the processor picks the next available sink via its configured selection mechanism. This implementation does not blacklist the failing sink and instead continues to optimistically attempt every available sink. If all sinks invocations result in failure, the selector propagates the failure to the sink runner.

If `backoff` is enabled, the sink processor will blacklist sinks that fail, removing them for selection for a given timeout. When the timeout ends, if the sink is still unresponsive timeout is increased exponentially to avoid potentially getting stuck in long waits on unresponsive sinks.

Required properties are in **bold**.

Property Name	Default	Description
processor.sinks	–	Space separated list of sinks that are participating in the group
processor.type	`default`	The component type name, needs to be `load_balance`
processor.backoff	true	Should failed sinks be backed off exponentially.
processor.selector	`round_robin`	Selection mechanism. Must be either `round_robin`, `random` or FQCN of custom class that inherits from `AbstractSinkSelector`
processor.selector.maxBackoffMillis	30000	used by backoff selectors to limit exponential backoff in miliseconds

Example for agent named a1:

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = load_balance
a1.sinkgroups.g1.processor.backoff = true
a1.sinkgroups.g1.processor.selector = random
```

Custom Sink Processor

Custom sink processors are not supported at the moment.

Event Serializers

The `file_roll` sink and the `hdfs` sink both support the `EventSerializer` interface. Details of the EventSerializers that ship with Flume are provided below.

Body Text Serializer

Alias: `text`. This interceptor writes the body of the event to an output stream without any transformation or modification. The event headers are ignored. Configuration options are as follows:

Property Name	Default	Description
appendNewline	true	Whether a newline will be appended to each event at write time. The default of true assumes that events do not contain newlines, for legacy reasons.

Example for agent named a1:

```
a1.sinks = k1
a1.sinks.k1.type = file_roll
a1.sinks.k1.channel = c1
a1.sinks.k1.sink.directory = /var/log/flume
a1.sinks.k1.sink.serializer = text
a1.sinks.k1.sink.serializer.appendNewline = false
```

Avro Event Serializer

Alias: `avro_event`. This interceptor serializes Flume events into an Avro container file. The schema used is the same schema used for Flume events in the Avro RPC mechanism. This serializers inherits from the `AbstractAvroEventSerializer` class. Configuration options are as follows:

Property Name	Default	Description
syncIntervalBytes	2048000	Avro sync interval, in approximate bytes.
compressionCodec	null	Avro compression codec. For supported codecs, see Avro's CodecFactory docs.

Example for agent named a1:

```
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
a1.sinks.k1.serializer = avro_event
a1.sinks.k1.serializer.compressionCodec = snappy
```

Flume Interceptors

Flume has the capability to modify/drop events in-flight. This is done with the help of interceptors. Interceptors are classes that implement `org.apache.flume.interceptor.Interceptor` interface. An interceptor can modify or even drop events based on any criteria chosen by the developer of the interceptor. Flume supports chaining of interceptors. This is made possible through by specifying the list of interceptor builder class names in the configuration. Interceptors are specified as a whitespace separated list in the source configuration. The order in which the interceptors are specified is the order in which they are invoked. The list of events returned by one interceptor is passed to the next interceptor in the chain. Interceptors can modify or drop events. If an interceptor needs to drop events, it just does not return that event in the list that it returns. If it is to drop all events, then it simply returns an empty list. Interceptors are named components, here is an example of how they are created through configuration:

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1
a1.sources.r1.interceptors = i1 i2
a1.sources.r1.interceptors.i1.type = org.apache.flume.interceptor.HostInterceptor$Builder
a1.sources.r1.interceptors.i1.preserveExisting = false
a1.sources.r1.interceptors.i1.hostHeader = hostname
a1.sources.r1.interceptors.i2.type = org.apache.flume.interceptor.TimestampInterceptor$Builder
a1.sinks.k1.filePrefix = FlumeData.%{CollectorHost}.%Y-%m-%d
a1.sinks.k1.channel = c1
```

Note that the interceptor builders are passed to the type config parameter. The interceptors are themselves configurable and can be passed configuration values just like they are passed to any other configurable component. In the above example, events are passed to the HostInterceptor first and the events returned by the HostInterceptor are then passed along to the TimestampInterceptor. You can specify either the fully qualified class name (FQCN) or the alias `timestamp`. If you have multiple collectors writing to the same HDFS path then you could also use the HostInterceptor.

Timestamp Interceptor

This interceptor inserts into the event headers, the time in millis at which it processes the event. This interceptor inserts a header with key `timestamp` whose value is the relevant timestamp. This interceptor can preserve an existing timestamp if it is already present in the configuration.

Property Name	Default	Description
type	–	The component type name, has to be `timestamp` or the FQCN

preserveExisting	false	If the timestamp already exists, should it be preserved - true or false

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.channels =  c1
a1.sources.r1.type = seq
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = timestamp
```

Host Interceptor

This interceptor inserts the hostname or IP address of the host that this agent is running on. It inserts a header with key `host` or a configured key whose value is the hostname or IP address of the host, based on configuration.

Property Name	Default	Description
type	–	The component type name, has to be `host`
preserveExisting	false	If the host header already exists, should it be preserved - true or false
useIP	true	Use the IP Address if true, else use hostname.
hostHeader	host	The header key to be used.

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = host
a1.sources.r1.interceptors.i1.hostHeader = hostname
```

Static Interceptor

Static interceptor allows user to append a static header with static value to all events.

The current implementation does not allow specifying multiple headers at one time. Instead user might chain multiple static interceptors each defining one static header.

Property Name	Default	Description
type	–	The component type name, has to be `static`
preserveExisting	true	If configured header already exists, should it be preserved - true or false
key	key	Name of header that should be created
value	value	Static value that should be created

Example for agent named a1:

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.channels =  c1
a1.sources.r1.type = seq
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = static
a1.sources.r1.interceptors.i1.key = datacenter
a1.sources.r1.interceptors.i1.value = NEW_YORK
```

Regex Filtering Interceptor

This interceptor filters events selectively by interpreting the event body as text and matching the text against a configured regular expression. The supplied regular expression can be used to include events or exclude events.

Property Name	Default	Description
type	–	The component type name has to be `regex_filter`
regex	".*"	Regular expression for matching against events
excludeEvents	false	If true, regex determines events to exclude, otherwise regex determines events to include.

Regex Extractor Interceptor

This interceptor extracts regex match groups using a specified regular expression and appends the match groups as headers on the event. It also supports pluggable serializers for formatting the match groups before adding them as event headers.

Property Name	Default	Description
type	–	The component type name has to be `regex_extractor`
regex	–	Regular expression for matching against events
serializers	–	Space-separated list of serializers for mapping matches to header names and serializing their values. (See example below) Flume provides built-in support for the following serializers: `org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer` `org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer`
serializers.<s1>.type	default	Must be `default` (org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer),

		`org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer`, or the FQCN of a custom class that implements `org.apache.flume.interceptor.RegexExtractorInterceptorSerializer`
serializers.<s1>.**name**	–	
serializers.*	–	Serializer-specific properties

The serializers are used to map the matches to a header name and a formatted header value, by default you only need to specify the header name and the default `org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer` will be used. This serializer simply maps the matches to the specified header name and passes the value through as it was extracted by the regex. You can plug custom serializer implementations into the extractor using the fully qualified class name (FQCN) to format the matches in anyway you like.

Example 1:

If the Flume event body contained `1:2:3.4foobar5` and the following configuration was used

```
agent.sources.r1.interceptors.i1.regex = (\\d):(\\d):(\\d)
agent.sources.r1.interceptors.i1.serializers = s1 s2 s3
agent.sources.r1.interceptors.i1.serializers.s1.name = one
agent.sources.r1.interceptors.i1.serializers.s2.name = two
agent.sources.r1.interceptors.i1.serializers.s3.name = three
```

The extracted event will contain the same body but the following headers will have been added `one=>1, two=>2, three=>3`

Example 2:

If the Flume event body contained `2012-10-18 18:47:57,614 some log line` and the following configuration was used

```
agent.sources.r1.interceptors.i1.regex = ^(?:\\n)?(\\d\\d\\d\\d-\\d\\d-\\d\\d\\s\\d\\d:\\d\\d)
agent.sources.r1.interceptors.i1.serializers = s1
agent.sources.r1.interceptors.i1.serializers.s1.type = org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer
agent.sources.r1.interceptors.i1.serializers.s1.name = timestamp
agent.sources.r1.interceptors.i1.serializers.s1.pattern = yyyy-MM-dd HH:mm
```

the extracted event will contain the same body but the following headers will have been added `timestamp=>1350611220000`

Flume Properties

Property Name	Default	Description
flume.called.from.service	–	If this property is specified then the Flume agent will continue polling for the config file even if the config file is not found at the expected location. Otherwise, the Flume agent will terminate if the config doesn't exist at the expected location. No property value is needed when setting this property (eg, just specifying -Dflume.called.from.service is enough)

Property: flume.called.from.service

Flume periodically polls, every 30 seconds, for changes to the specified config file. A Flume agent loads a new configuration from the config file if either an existing file is polled for the first time, or if an existing file's modification date has changed since the last time it was polled. Renaming or moving a file does not change its modification time. When a Flume agent polls a non-existent file then one of two things happens: 1. When the agent polls a non-existent config file for the first time, then the agent behaves according to the flume.called.from.service property. If the property is set, then the agent will continue polling (always at the same period – every 30 seconds). If the property is not set, then the agent immediately terminates. ...OR... 2. When the agent polls a non-existent config file and this is not the first time the file is polled, then the agent makes no config changes for this polling period. The agent continues polling rather than terminating.

Log4J Appender

Appends Log4j events to a flume agent's avro source. A client using this appender must have the flume-ng-sdk in the classpath (eg, flume-ng-sdk-1.3.0.jar). Required properties are in **bold**.

Property Name	Default	Description
Hostname	–	The hostname on which a remote Flume agent is running with an avro source.
Port	–	The port at which the remote Flume agent's avro source is listening.

Sample log4j.properties file:

```
#...
log4j.appender.flume = org.apache.flume.clients.log4jappender.Log4jAppender
log4j.appender.flume.Hostname = example.com
log4j.appender.flume.Port = 41414

# configure a class's logger to output to the flume appender
log4j.logger.org.example.MyClass = DEBUG,flume
#...
```

Security

The HDFS sink supports Kerberos authentication if the underlying HDFS is running in secure mode. Please refer to the HDFS Sink section for configuring the HDFS sink Kerberos-related options.

Monitoring

Monitoring in Flume is still a work in progress. Changes can happen very often. Several Flume components report metrics to the JMX platform MBean server. These metrics can be queried using Jconsole.

Ganglia Reporting

Flume can also report these metrics to Ganglia 3 or Ganglia 3.1 metanodes. To report metrics to Ganglia, a flume agent must be started with this support. The Flume agent has to be started by passing in the following parameters as system properties prefixed by `flume.monitoring.`, and can be specified in the flume-env.sh:

Property Name	Default	Description
type	–	The component type name, has to be `ganglia`
hosts	–	Comma separated list of `hostname:port`
pollInterval	60	Time, in seconds, between consecutive reporting to ganglia server
isGanglia3	false	Ganglia server version is 3. By default, Flume sends in ganglia 3.1 format

We can start Flume with Ganglia support as follows:

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.monitoring.type=ganglia -Dflume.monitoring.hosts=com.example:1234,com.
```

JSON Reporting

Flume can also report metrics in a JSON format. To enable reporting in JSON format, Flume hosts a Web server on a configurable port. Flume reports metrics in the following JSON format:

```
{
"typeName1.componentName1" : {"metric1" : "metricValue1", "metric2" : "metricValue2"},
"typeName2.componentName2" : {"metric3" : "metricValue3", "metric4" : "metricValue4"}
}
```

Here is an example:

```
{
"CHANNEL.fileChannel":{"EventPutSuccessCount":"468085",
                        "Type":"CHANNEL",
                        "StopTime":"0",
                        "EventPutAttemptCount":"468086",
                        "ChannelSize":"233428",
                        "StartTime":"1344882233070",
                        "EventTakeSuccessCount":"458200",
                        "ChannelCapacity":"600000",
                        "EventTakeAttemptCount":"458288"},
"CHANNEL.memChannel":{"EventPutSuccessCount":"22948908",
                     "Type":"CHANNEL",
                     "StopTime":"0",
                     "EventPutAttemptCount":"22948908",
                     "ChannelSize":"5",
                     "StartTime":"1344882209413",
                     "EventTakeSuccessCount":"22948900",
                     "ChannelCapacity":"100",
                     "EventTakeAttemptCount":"22948908"}
}
```

Property Name	Default	Description
type	–	The component type name, has to be `http`
port	41414	The port to start the server on.

We can start Flume with Ganglia support as follows:

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.monitoring.type=http -Dflume.monitoring.port=34545
```

Metrics will then be available at **http://<hostname>:<port>/metrics** webpage. Custom components can report metrics as mentioned in the Ganglia section above.

Custom Reporting

It is possible to report metrics to other systems by writing servers that do the reporting. Any reporting class has to implement the interface, `org.apache.flume.instrumentation.MonitorService`. Such a class can be used the same way the GangliaServer is used for reporting. They can poll the platform mbean server to poll the mbeans for metrics. For example, if an HTTP monitoring service called `HTTPReporting` can be used as follows:

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.monitoring.type=com.example.reporting.HTTPReporting -Dflume.monitoring
```

Property Name	Default	Description
type	–	The component type name, has to be FQCN

Reporting metrics from custom components

Any custom flume components should inherit from the `org.apache.flume.instrumentation.MonitoredCounterGroup` class. The class should then provide getter methods for each of the metrics it exposes. See the code below. The MonitoredCounterGroup expects a list of attributes whose metrics are exposed by this class. As of now, this class only supports exposing metrics as long values.

```
public class SinkCounter extends MonitoredCounterGroup implements
    SinkCounterMBean {

  private static final String COUNTER_CONNECTION_CREATED =
    "sink.connection.creation.count";

  private static final String COUNTER_CONNECTION_CLOSED =
    "sink.connection.closed.count";

  private static final String COUNTER_CONNECTION_FAILED =
    "sink.connection.failed.count";

  private static final String COUNTER_BATCH_EMPTY =
    "sink.batch.empty";

  private static final String COUNTER_BATCH_UNDERFLOW =
      "sink.batch.underflow";

  private static final String COUNTER_BATCH_COMPLETE =
    "sink.batch.complete";

  private static final String COUNTER_EVENT_DRAIN_ATTEMPT =
    "sink.event.drain.attempt";

  private static final String COUNTER_EVENT_DRAIN_SUCCESS =
    "sink.event.drain.sucess";

  private static final String[] ATTRIBUTES = {
    COUNTER_CONNECTION_CREATED, COUNTER_CONNECTION_CLOSED,
    COUNTER_CONNECTION_FAILED, COUNTER_BATCH_EMPTY,
    COUNTER_BATCH_UNDERFLOW, COUNTER_BATCH_COMPLETE,
    COUNTER_EVENT_DRAIN_ATTEMPT, COUNTER_EVENT_DRAIN_SUCCESS
  };


  public SinkCounter(String name) {
    super(MonitoredCounterGroup.Type.SINK, name, ATTRIBUTES);
  }

  @Override
  public long getConnectionCreatedCount() {
    return get(COUNTER_CONNECTION_CREATED);
  }

  public long incrementConnectionCreatedCount() {
    return increment(COUNTER_CONNECTION_CREATED);
  }

}
```

Topology Design Considerations

Flume is very flexible and allows a large range of possible deployment scenarios. If you plan to use Flume in a large, production deployment, it is prudent to spend some time thinking about how to express your problem in terms of a Flume topology. This section covers a few considerations.

Is Flume a good fit for your problem?

If you need to ingest textual log data into Hadoop/HDFS then Flume is the right fit for your problem, full stop. For other use cases, here are some guidelines:

Flume is designed to transport and ingest regularly generated event data over relatively stable, potentially complex topologies. The notion of "event data" is very broadly defined. To Flume, an event is just a generic blob of bytes. There are some limitations on how large an event can be - for instance, it cannot be larger than you can store in memory or on disk on a single machine - but in practice flume events can be everything from textual log entries to image files. The key property of an event is that they are generated in a continuous, streaming fashion. If your data is not regularly generated (i.e. you are trying to do a single bulk load of data into a Hadoop cluster) then Flume will still work, but it is probably overkill for your situation. Flume likes relatively stable topologies. Your topologies do not need to be immutable, because Flume can deal with changes in topology without losing data and can also tolerate periodic reconfiguration due to fail-over or provisioning. It probably won't work well if you plant to change topologies every day, because reconfiguration takes some thought and overhead.

Flow reliability in Flume

The reliability of a Flume flow depends on several factors. By adjusting these factors, you can achieve a wide array of reliability options with Flume.

What type of channel you use. Flume has both durable channels (those which will persist data to disk) and non durable channels (those which will lose data if a machine fails). Durable channels use disk-based storage, and data stored in such channels will persist across machine restarts or non disk-related failures.

Whether your channels are sufficiently provisioned for the workload. Channels in Flume act as buffers at various hops. These buffers have a fixed capacity, and once that capacity is full you will create back pressure on earlier points in the flow. If this pressure propagates to the source of the flow, Flume will become unavailable and may lose data.

Whether you use redundant topologies. Flume let's you replicate flows across redundant topologies. This can provide a very easy source of fault tolerance and one which is overcomes both disk or machine failures.

The best way to think about reliability in a Flume topology is to consider various failure scenarios and their outcomes. What happens if a disk fails? What happens if a machine fails? What happens if your terminal sink (e.g. HDFS) goes down for some time and you have back pressure? The space of possible designs is huge, but the underlying questions you need to ask are just a handful.

Flume topology design

The first step in designing a Flume topology is to enumerate all sources and destinations (terminal sinks) for your data. These will define the edge points of your topology. The next consideration is whether to introduce intermediate aggregation tiers or event routing. If you are collecting data form a large number of sources, it can be helpful to aggregate the data in order to simplify ingestion at the terminal sink. An aggregation tier can also smooth out burstiness from sources or unavailability at sinks, by acting as a buffer. If you are routing data between different locations, you may also want to split flows at various points: this creates sub-topologies which may themselves include aggregation points.

Sizing a Flume deployment

Once you have an idea of what your topology will look like, the next question is how much hardware and networking capacity is needed. This starts by quantifying how much data you generate. That is not always a simple task! Most data streams are bursty (for instance, due to diurnal patterns) and potentially unpredictable. A good starting point is to think about the maximum throughput you'll have in each tier of the topology, both in terms of *events per second* and *bytes per second*. Once you know the required throughput of a given tier, you can calulate a lower bound on how many nodes you require for that tier. To determine attainable throughput, it's best to experiment with Flume on your hardware, using synthetic or sampled event data. In general, disk-based channels should get 10's of MB/s and memory based channels should get 100's of MB/s or more. Performance will vary widely, however depending on hardware and operating environment.

Sizing aggregate throughput gives you a lower bound on the number of nodes you will need to each tier. There are several reasons to have additional nodes, such as increased redundancy and better ability to absorb bursts in load.

Troubleshooting

Handling agent failures

If the Flume agent goes down then the all the flows hosted on that agent are aborted. Once the agent is restarted, then flow will resume. The flow using file channel or other stable channel will resume processing events where it left off. If the agent can't be restarted on the same, then there an option to migrate the database to another hardware and setup a new Flume agent that can resume processing the events saved in the db. The database HA futures can be leveraged to move the Flume agent to another host.

Compatibility

HDFS

Currently Flume supports HDFS 0.20.2 and 0.23.

AVRO

TBD

Additional version requirements

TBD

Tracing

TBD

More Sample Configs

TBD

Component Summary

Component Interface	Type Alias	Implementation Class
org.apache.flume.Channel	memory	org.apache.flume.channel.MemoryChannel
org.apache.flume.Channel	jdbc	org.apache.flume.channel.jdbc.JdbcChannel
org.apache.flume.Channel	recoverablememory	org.apache.flume.channel.recoverable.memory.RecoverableMemory
org.apache.flume.Channel	file	org.apache.flume.channel.file.FileChannel

org.apache.flume.Channel	–	org.apache.flume.channel.PseudoTxnMemoryChannel
org.apache.flume.Channel	–	org.example.MyChannel
org.apache.flume.Source	avro	org.apache.flume.source.AvroSource
org.apache.flume.Source	netcat	org.apache.flume.source.NetcatSource
org.apache.flume.Source	seq	org.apache.flume.source.SequenceGeneratorSource
org.apache.flume.Source	exec	org.apache.flume.source.ExecSource
org.apache.flume.Source	syslogtcp	org.apache.flume.source.SyslogTcpSource
org.apache.flume.Source	multiport_syslogtcp	org.apache.flume.source.MultiportSyslogTCPSource
org.apache.flume.Source	syslogudp	org.apache.flume.source.SyslogUDPSource
org.apache.flume.Source	–	org.apache.flume.source.avroLegacy.AvroLegacySource
org.apache.flume.Source	–	org.apache.flume.source.thriftLegacy.ThriftLegacySource
org.apache.flume.Source	–	org.example.MySource
org.apache.flume.Sink	null	org.apache.flume.sink.NullSink
org.apache.flume.Sink	logger	org.apache.flume.sink.LoggerSink
org.apache.flume.Sink	avro	org.apache.flume.sink.AvroSink
org.apache.flume.Sink	hdfs	org.apache.flume.sink.hdfs.HDFSEventSink
org.apache.flume.Sink	–	org.apache.flume.sink.hbase.HBaseSink
org.apache.flume.Sink	–	org.apache.flume.sink.hbase.AsyncHBaseSink
org.apache.flume.Sink	–	org.apache.flume.sink.elasticsearch.ElasticSearchSink
org.apache.flume.Sink	file_roll	org.apache.flume.sink.RollingFileSink
org.apache.flume.Sink	irc	org.apache.flume.sink.irc.IRCSink
org.apache.flume.Sink	–	org.example.MySink
org.apache.flume.ChannelSelector	replicating	org.apache.flume.channel.ReplicatingChannelSelector
org.apache.flume.ChannelSelector	multiplexing	org.apache.flume.channel.MultiplexingChannelSelector
org.apache.flume.ChannelSelector	–	org.example.MyChannelSelector
org.apache.flume.SinkProcessor	default	org.apache.flume.sink.DefaultSinkProcessor
org.apache.flume.SinkProcessor	failover	org.apache.flume.sink.FailoverSinkProcessor
org.apache.flume.SinkProcessor	load_balance	org.apache.flume.sink.LoadBalancingSinkProcessor
org.apache.flume.SinkProcessor	–	
org.apache.flume.interceptor.Interceptor	timestamp	org.apache.flume.interceptor.TimestampInterceptor$Builder
org.apache.flume.interceptor.Interceptor	host	org.apache.flume.interceptor.HostInterceptor$Builder
org.apache.flume.interceptor.Interceptor	static	org.apache.flume.interceptor.StaticInterceptor$Builder
org.apache.flume.interceptor.Interceptor	regex_filter	org.apache.flume.interceptor.RegexFilteringInterceptor$Builder
org.apache.flume.interceptor.Interceptor	regex_extractor	org.apache.flume.interceptor.RegexFilteringInterceptor$Builder
org.apache.flume.channel.file.encryption.KeyProvider$Builder	jceksfile	org.apache.flume.channel.file.encryption.JCEFileKeyProvider
org.apache.flume.channel.file.encryption.KeyProvider$Builder	–	org.example.MyKeyProvider
org.apache.flume.channel.file.encryption.CipherProvider	aesctrnopadding	org.apache.flume.channel.file.encryption.AESCTRNoPaddingProvic
org.apache.flume.channel.file.encryption.CipherProvider	–	org.example.MyCipherProvider
org.apache.flume.serialization.EventSerializer$Builder	text	org.apache.flume.serialization.BodyTextEventSerializer$Builder
org.apache.flume.serialization.EventSerializer$Builder	avro_event	org.apache.flume.serialization.FlumeEventAvroEventSerializer$Buil
org.apache.flume.serialization.EventSerializer$Builder	–	org.example.MyEventSerializer$Builder

Alias Conventions

These conventions for alias names are used in the component-specific examples above, to keep the names short and consistent across all examples.

Alias Name	Alias Type
a	**a**gent
c	**c**hannel
r	sou**r**ce
k	sin**k**
g	sink **g**roup
i	**i**nterceptor
y	ke**y**
h	**h**ost
s	**s**erializer